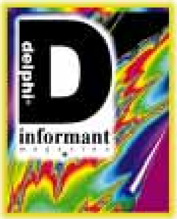


On the Cover

Delphi / Object Pascal

By *Robert Vivrette*

Face Value

Creating an Attractive and Useful Interface

First impressions are important. What your application says to the user is going to reflect positively or negatively on your ability to develop a functional and professional product.

This article will address issues of how to develop a professional application. Some of the topics we'll cover may seem obvious — some may even seem trivial. However, the tips and techniques presented form at least a basic set of quality standards that programmers today should consider when developing applications.

To illustrate some of the principles that we'll cover in this article, I have written a handy utility called BMPVIEW (see Figure 1) that displays all the bitmap files (.BMP) in a directory. Although it might seem a mundane application, it illustrates a number of important elements of program design.

Be Considerate of the Environment

These days it can almost be said that no two PCs are the same. As a result, your application should make as few assumptions about the user's machine as possible. However, it is appropriate to make some threshold assumptions, such as "the PC must be running Windows 3.1 or higher, and should be a 386DX/33 or better to run properly." The less you assume beyond that, the better.

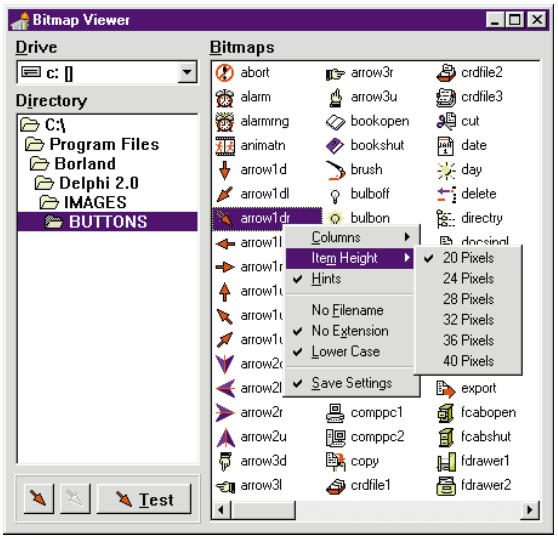


Figure 1: The BMPVIEW program.

For example, your program should not assume a certain screen size or color depth. As much as possible, different screen resolutions should be tested. Screen resolution could vary from 640x480 up to 1280x1024, or higher. If you design a program to look good at lower resolutions, it might wind up resembling something the size of a postage stamp at higher resolutions.

In addition, PCs can have different color depths. Generally, users are running their machines in 16 or 256 color configurations. Yet some users may be running thousands (or millions) or colors. Here are some questions to consider:

- What impact will these changes have on your application?
- Are you using 256-color images that will look hideous on a 16-color machine? If this is the case, should you include 16-color versions of the graphics and have your application adapt?
- If the program must have 256 colors or better, does it test for it, then provide the user with a helpful explanation of what must be done to use the program to its full capability?

In addition, you should test to see what happens when you run other programs concurrently. Keep in mind that Windows must manage the palette of colors used by every application, and when you switch between programs, the visual effect could be unpleasant.

Something that may be more difficult to test would be the effect of running an application on a machine with a gray-scale monitor. Will your selected colors be too subtle to discern when viewed in shades of gray? If you have the luxury of knowing that all users of the application will have color displays, you can dismiss this issue.

Now, designing your program to *tolerate* certain resolutions or colors is one thing, but will it take *advantage* of them?

UI Aerobics

Every time I evaluate a program, I put it through a brief regimen of calisthenics. I resize the application window with every edge, maximize it, minimize it, even see how large or small I can get it. What if the user shrinks the application window — will the edges obscure controls? Many novice users may be stumped by the appearance this might give. Alternatively, if you expand the program to fill the screen, what happens? Will all the controls retain their location and the rest of the form contain nothing but empty space? A well-designed application should adapt and take advantage of the extended screen space, moving and expanding the size of list boxes, edit boxes, memo fields, and the like.

The BMPVIEW program does just that. It can dynamically adjust its controls to handle any screen resolution on which it might be run. The key to this is to use the *TForm Resize* event (see Figure 2).

Whenever the main window of the program is resized, I make certain adjustments to the controls. The first is to move the two buttons in the bottom left of the screen. I move them to a point that is six pixels up from the bottom edge (*ClientHeight*) of the main form. Since *ClientHeight* is changing during the resize, the buttons will maintain their distance from the bottom edge.

```
procedure TBitmapViewerDialog.FormResize(Sender: TObject);
begin
  { Align the Samples panel to the bottom left }
  with SamplesPanel do
    SetBounds(Left,Self.ClientHeight-Height-6,
              Width,Height);
  { The DirList should take up all space
    remaining above }
  DirList.Height := SamplesPanel.Top-DirList.Top-6;
  { Perform Update to ensure all controls paint before
    the BmpList }
  Update;
  { BmpList should take up all space remaining on form }
  with BmpList do
    SetBounds(Left,Top,Self.ClientWidth-Left-6,
              Self.ClientHeight-Top-6);
end;
```

Figure 2: The *Resize* event of *TForm*.

Next, I adjust the directory list control to stop six pixels above the top edge of one of the buttons (which we just moved). We are not adjusting the width or placement of the directory list control. Thus, only its height will change.

Last, I use the remaining space for the bitmap viewer list box. I want to keep its upper left corner the same, but I want the width and height to be six pixels short of the bottom right corner. Delphi provides the ability to do most of this automatically for most controls by means of the *Align* property. However, I have found that there are many behaviors that *Align* cannot duplicate, so I often revert to using the *Resize* method.

If you're assuming the program will always be the same size, you should set the form's *BorderStyle* property to *bsSingle* to prevent resizing. Better yet, why not use the *bsDialog* style so that it will have a more attractive 3D look?

If you want to allow resizing — but only within certain limits — you should have the program respond to the WM_GETMINMAXINFO message. Whenever a user resizes a form, Windows sends one of these messages. The response it receives determines the limits of how large or how small the form will be permitted to become. The nice part about this is that it is done *as the form is being resized*. By setting a minimum width and height in this way, Windows allows the user to reduce the form's size only until this limit is reached. Then the form's edge stops moving.

Again, the BMPVIEW program uses one of these message handlers. There is no point allowing the user to resize the application below a usable size. When the form shrinks past a certain point, the bitmap list box and/or the directory list box will disappear, making the application non-functional until it's enlarged. The following message handler limits the form's size to 300 pixels wide and 250 pixels high:

```
procedure TForm1.WMGetMinMaxInfo(
  var Message : TWMGetMinMaxInfo);
begin
  with Message.MinMaxInfo^ do begin
    ptMinTrackSize.X := 300;   { Minimum width }
    ptMinTrackSize.Y := 250;   { Minimum height }
  end;
  Message.Result := 0;
  inherited;
end;
```

With the dynamic moving and sizing of the main form and its controls — as well as the low-end limit to the form's size — it can truly take advantage of any kind of screen real estate that we throw at it. The BMPVIEW program extends these capabilities by allowing the user to modify the number of columns displayed as well as the height of each of the items in the bitmap list box.

Figure 3 shows how the program might look after a user resizes it.

Look for Annoyances

Do you see any annoying things in the Delphi IDE? Not

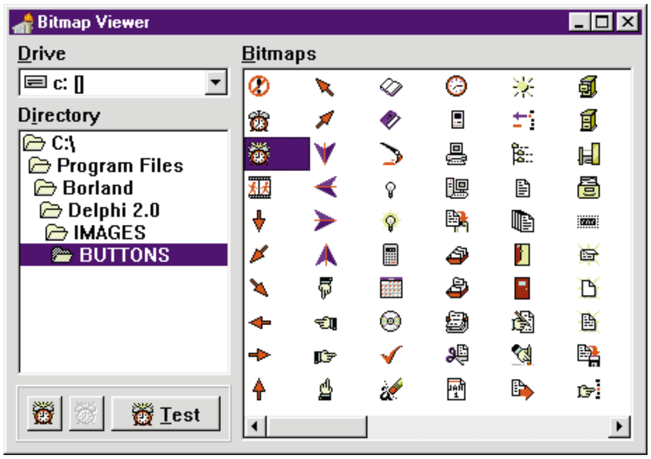


Figure 3: Configuration settings allow the appearance and behavior of BMPVIEW to be changed. Here, you can see that six columns of bitmaps are shown, without file name extensions. Notice the abbreviated length of this figure when compared to Figure 1.

many I'll bet, and there's a good reason. Since Delphi was written with Delphi, its developers and programmers had to use elements of it (such as the IDE) every day. Since they had to use it constantly, they didn't tolerate little functional annoyances. The product is, of course, better as a result. They wouldn't put up with these problems, and they knew we wouldn't either.

When developing your application, keep this same attitude. If you see an element of the program that is annoying or unnecessarily difficult, fix it. Don't think "Well, I'll fix it in version 1.1." Just because you'll allow these annoyances does not mean that your users will. And they may not let you get to 1.1!

What kind of annoyances am I talking about? Let's address a few that come to mind.

Save Configuration Information. This is one of my biggest gripes. Whenever your program ends, it should write certain information to an .INI file so it can restore these defaults the next time it runs. This would include things such as form size and placement; default directories for common dialog boxes (e.g. Open and Save dialog boxes); overridden filters in common dialog boxes (e.g. the wild-card specifications for an Open dialog box), etc.

These settings should not be written to the WIN.INI file (unless there is a specific reason to do so). Rather, they should be written to a private .INI file or, better yet, to the System Registry in Windows 95. For this simple application, I'll be writing these settings to a private .INI file called BMPVIEW.INI that Delphi will automatically place in the \WINDOWS directory.

The BMPVIEW program makes full use of the *TIniFile* object to read and write configuration information. It remembers the last directory it was in, the window size and placement, as well as the various configuration settings within the program (such as the number of columns to display, or the height of each item in the bitmap list box). The procedures shown in Figure 4 control the loading and saving of this information.

The *ReadSettings* method is called when the form is created. This loads the information from the .INI file and sets the appropriate values of the various application properties. The *WriteSettings* method does just the reverse; it writes the current state of these values so the next time the program is executed, it will start with the same settings.

A Logical Tab Order for Controls. Make sure the tab order has been set and that it's appropriate. A user should be able to anticipate where the focus will move if Tab is pressed. Test this with a user who is unfamiliar with the application.

Sometimes it may be appropriate for you to move the focus for the user. For example, if you know that after clicking on a particular button the user will need to fill in an edit field, why not shift the focus there immediately so the user can begin typing?

Support for the Keyboard. This seems to be a commonly overlooked topic. Before you ship your program, ask yourself, "Without a mouse, could the user effectively navigate the application?" The answer should always be "Yes." It's a common thing to overlook, but many developers assume that the mouse will always be available. Not a good assumption! What if your user is running the program on a laptop and forgot the mouse? Would the application be unusable or difficult to use?

I test this by placing the mouse out of reach and then putting the program through its paces. Command buttons and menus should have hot-key shortcuts and the tab sequence should cycle through all appropriate controls. Make a note of all program functions that cannot be accessed and add keyboard support for them.

In the BMPVIEW program, I included a pop-up menu over the **Bitmaps** list box to control the number of columns to display, the height of each item, and the method of loading and displaying bitmaps (see Figure 5). It was necessary to include a way of activating this pop-up menu using the keyboard. This was done by turning on the main form's *KeyPreview* property, and then modifying its *OnKeyDown* method as follows:

```
procedure TForm1.FormKeyDown(Sender: TObject;
  var Key: Word; Shift: TShiftState);
begin
  if (Shift = [ssCtrl]) and (Key = VK_1) then
    PopUpMenu1.PopUp(Left+ BmpList.Left+
                   (BmpList.Width div 2),
                    Top+BmpList.Top+Height-ClientHeight+
                   (BmpList.Height div 2));
end;
```

This procedure instructs the form to activate the pop-up menu in the middle of the **Bitmaps** list area when the user presses Ctrl 1.

Overuse of Hints. Delphi's Help Hint feature is wonderful, and it takes little or no programming to implement a fully functional hint system in your program. However, just because it's easy to include does not mean that you should add this fea-

```
procedure TBitmapViewerDialog.ReadSettings;
begin
  { Read all configuration and window settings from the INI file }
  SettingsIni := TIniFile.Create('BMPVIEW.INI');
  { Configuration Settings }
  DirList.Directory      :=
    SettingsIni.ReadString('Options','Directory','C:\');
  BmpList.Columns        :=
    SettingsIni.ReadInteger('Options','Columns',2);
  BmpList.ItemHeight     :=
    SettingsIni.ReadInteger('Options','ItemHeight',24);
  pmHints.Checked        :=
    SettingsIni.ReadBool('Options','Hints',True);
  pmSaveSettings.Checked :=
    SettingsIni.ReadBool('Options','SaveSettings',True);
  pmLowerCase.Checked    :=
    SettingsIni.ReadBool('Options','LowerCase',False);
  pmNoFilename.Checked   :=
    SettingsIni.ReadBool('Options','NoFilename',False);
  pmNoExtension.Checked  :=
    SettingsIni.ReadBool('Options','NoExtension',True);
  { Window position settings }
  FormL := SettingsIni.ReadInteger('Window','Left',   -99);
  FormT := SettingsIni.ReadInteger('Window','Top',    -99);
  FormW := SettingsIni.ReadInteger('Window','Width',  -99);
  FormH := SettingsIni.ReadInteger('Window','Height', -99);
  FormState := SettingsIni.ReadInteger('Window','State',0);
  { Perform SetBounds only if all parameters have
    been specified }
  if (FormL >= 0) and (FormT >= 0) and
     (FormW > 0)  and (FormH > 0) then
    SetBounds(FormL,FormT,FormW,FormH);
  { Set the previous window state }
  case FormState of
    0 : WindowState := wsNormal;
    1 : WindowState := wsMaximized;
    2 : WindowState := wsMinimized;
  end;
  { Set initial menu check states }
  SetColumnCheck(BmpList.Columns);
  SetPixelCheck(BmpList.ItemHeight);
  SetHintsCheck;
```

```
    { Free the TIniFile object }
  SettingsIni.Free;
end;

procedure TBitmapViewerDialog.WriteSettings;
begin
  { Write out configuration & Window settings to INI file }
  SettingsIni := TIniFile.Create('BMPVIEW.INI');
  SettingsIni.WriteBool('Options','SaveSettings',
                        pmSaveSettings.Checked);
  { Only save remaining items if SaveSettings is checked }
  if pmSaveSettings.Checked then
    begin
      SettingsIni.WriteString(
        'Options','Directory',DirList.Directory);
      SettingsIni.WriteInteger(
        'Options','Columns',BmpList.Columns);
      SettingsIni.WriteInteger(
        'Options','ItemHeight',BmpList.ItemHeight);
      SettingsIni.WriteBool(
        'Options','Hints',pmHints.Checked);
      SettingsIni.WriteBool(
        'Options','LowerCase',pmLowerCase.Checked);
      SettingsIni.WriteBool(
        'Options','NoFilename',pmNoFilename.Checked);
      SettingsIni.WriteBool(
        'Options','NoExtension',pmNoExtension.Checked);
      SettingsIni.WriteInteger('Window','Left',  Left);
      SettingsIni.WriteInteger('Window','Top',   Top);
      SettingsIni.WriteInteger('Window','Width', Width);
      SettingsIni.WriteInteger('Window','Height',Height);
           case WindowState of
         wsNormal    :
           SettingsIni.WriteInteger('Window','State',0);
         wsMaximized :
           SettingsIni.WriteInteger('Window','State',1);
         wsMinimized :
           SettingsIni.WriteInteger('Window','State',2);
      end;
    end;
    SettingsIni.Free;
end;
```

Figure 4: The *ReadSettings* and *WriteSettings* procedures are responsible for reading and writing information from and to the application's .INI file.

ture. I've seen programs where every on-screen control has an associated hint. This gets annoying after about 10 minutes.

The recommendation, therefore, is to only include hints on things where it's necessary. If there are a lot of hints, you might want to have a configuration option to allow the user to enable or disable them. (And, in light of our first annoyance mentioned above, you're going to save the hint status to an .INI file, aren't you?)

Do You Feel Like You Look?

It's an overly-used phrase, but there is much to be said for a program's "look and feel." As stated at the beginning of this article, a user's first impression of an application is important. With that in mind, let's look at a few things that affect a program's "look and feel."

Use of Color. Windows applications inherit much of their color settings from the operating system itself. When selecting colors for forms, buttons, etc., be careful when using hard-coded color references such as *clGreen* or *clRed*. Instead, use appropriate constants such as *clButtonFace*, or *clWindow*.

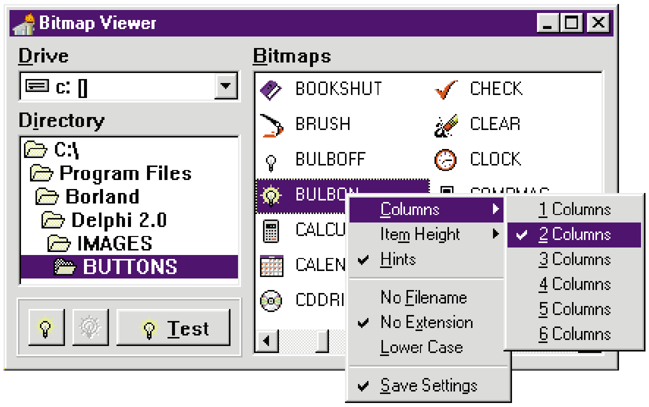


Figure 5: Another view of the BMPVIEW program. The user configured the **Bitmaps** listbox to enable hints, display two columns of bitmaps, omit the file name extension, and save settings on exit. For ease of use and versatility, these settings can also be configured via the keyboard.

This way, your application will fit in with the color settings that the user has defined from within Windows, and can still look good after the user selects a different setting.

Use of Fonts. You just purchased that cool set of 800 typefaces and are all set to jazz up your applications. Yes, these new fonts will work on your applications as long they continue to run on your machine. However, once the program is moved to another computer, these fonts will probably no longer be available.

In this situation, Windows will substitute standard fonts for your fancy ones, which will probably mess up all your careful positioning of labels and controls on the form. To avoid this, stick with fonts that you know will be on all users' machines, such as MS Sans Serif, System, Arial, Courier, Times New Roman, or Small Fonts (to name a few). This way, the fonts your program uses will be the same on any machine.

Placement of Controls. Make sure the program's layout is neat and organized. Consider all these questions:

1) Are all controls placed in a logical arrangement? Are commonly used sets of controls located near each other to minimize mouse travel? Controls that are supposed to be in a row should be exactly on the same pixel row.
2) Does the form have an even margin around the edges? Controls that are jammed against the side of a form show that you didn't care enough to place them correctly. There should also be an even distribution of space between controls.
3) Do buttons have glyphs attached to them? Plain buttons with just text are dull and flat looking. Spruce up the buttons by using the BitButton controls and assigning appropriate glyphs to them.

By paying careful attention to the details that increase your application's usability, and by allowing the user to customize settings in the program, you'll find that user acceptance will come more easily.

The Sample Application

Figure 1 shows the BMPVIEW program in one configuration. By changing a few of the configuration settings (e.g. disabling file names, and increasing the number of columns displayed), the application can now look more like Figure 3.

When I created this program, I thought it would make an excellent replacement for the bitmap property editor in the Delphi IDE. Once installed in the IDE, this replacement allows the developer to click on *Glyph* properties for SpeedButtons or BitButtons and use this property editor to select an appropriate bitmap.

The change-over from bitmap viewer to property editor is actually quite simple (see end of article for download information). This project is featured in Chapter 5 of a book I co-authored, entitled *Delphi In Depth* [Osborne/McGraw-Hill, 1996]. If you would like to learn more about how BMPVIEW the bitmap viewer became BMPPROP the property editor, it's covered in detail in the book.

Conclusion

The way your program is put together reflects the kind of programmer you are. Hopefully the material we have covered here will serve to show some of the concepts and practices that go into the construction of a professional, intuitive, and attractive Windows application.

It's often the little things you do to a program that will have the greatest impact. The best part is that you aren't alone. With the help of the Delphi programming environment, it's surprisingly easy to achieve all of this, and much more. Δ

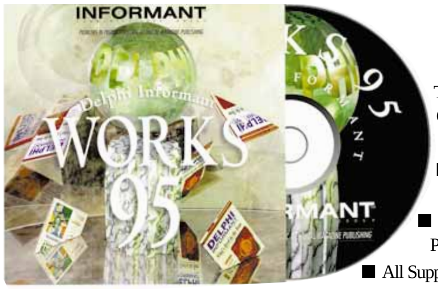
The demonstration projects referenced in this article are available on the Delphi Informant Works CD located in INFORM\96\JUL\DI9607RV.ZIP.

Robert Vivrette is a contract programmer for Pacific Gas & Electric and Technical Editor for *Delphi Informant*. He has worked as a game designer and computer consultant, and has experience in a number of programming languages. He can be reached on CompuServe at 76416,1373.

table of CONTENTS

The Must Have Reference Source For The Serious Delphi™ Developer

The Entire Text of Over 60 Technical Articles Appearing in Delphi™ Informant® in 1995



The Delphi™ Informant® Works 1995 CD-ROM Includes:

- Over 60 Technical Articles
- Text and Keyword Search Capability
- Improved Speed and Performance
- All Supporting Code and Sample Files
- 16-Page Color Booklet
- Electronic Version of *Delphi Power Tools* Catalog
- Electronic Version of *Borland C++ Power Tools* Catalog
- Third-Party Add-In Product Demos
- CompuServe Starter Kit with $15 Usage Credit.

A $105 Value Available Now for only

$39.95

California residents add 7¼% Sales Tax,
plus $5 shipping & handling for US orders.
(International orders add $15 shipping & handling)

Call Now Toll Free 1-800-88-INFORM

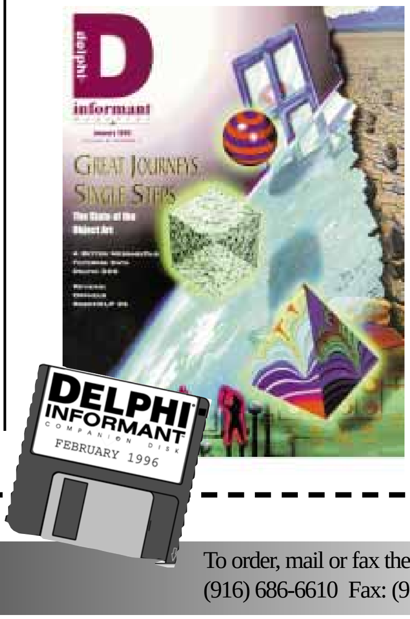
1-800-884-6367 Ask for offer # PI95 To order by mail,
send check or Money Order to:
Informant Communications Group, Inc.
ATTN: Works CD offer # PI95
10519 E. Stockton Blvd, Suite 142 Elk Grove, CA 95624-9704
or Fax your order to
916-686-8497

Get Informed!

Subscribe to Delphi Informant, The Complete Monthly Guide to Delphi Development and stay ahead of the rapid application development curve.

Order Now and Get One Issue FREE!

For a limited time you can receive the first issue FREE plus 12 additional issues for only $49.95 That's nearly 25% off the yearly cover price!



Each big issue of *Delphi Informant* is packed with Delphi tips, techniques , news, and more!

- Client/Server Application Development
- Delphi Programming
- Using the Borland Database Engine
- Object -Oriented Programming
- Creating Reusable Components
- Product Reviews
- News from the Delphi Community
- Delphi User Group Information

Payment Method...

□ Check (payable to Informant Communications Group)

□ Purchase Order--
Provide Number ____________

□ Visa

□ Mastercard

□ American Express

Card Number ____________

Expiration Date ____________

Signature ____________

International rates

Magazine-only

$54.95/year to Canada
$74.95/year to Mexico
$79.95/year to all other countries

Magazine AND Companion Disk Subscriptions

$124.95/year to Canada
$154.95/year to Mexico
$179.95 to all other countries

California Residents add 7¼% sales tax on disk subscription

□ **YES**

To order, mail or fax the form below or call
(916) 686-6610 Fax: (916) 686-8497

I want to sharpen my Delphi Programming skills. I've checked the subscription plan I'm interested in below

□ Magazine-Only Subscription Plan...
13 Issues Including One Bonus Issue at $49.95.

□ Magazine AND Companion Disk Subscription Plan...
13 Issues and Disks Including One Bonus Issue and Disk at $119.95
The Delphi Informant Companion Disk contains source code, support files, examples, utilities, samples, and more!

□ Delphi Informant Works 1995 CD-ROM $39.95
US residents add $5 shipping and handling. International customers add $15 shipping and handling.

Name ____________

Company ____________

Address ____________

City ____________ State ______ Zip Code ______

Country ____________ Phone ____________

FAX ____________ e-Mail ____________

PI95